Intro to C

Hello World

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

`#include`: preprocessor inserts `stdio.h` contents
`stdio.h`: contains `printf` declaration
`main`: program starts here
`void`: keyword for argument absence
`{ }`: basic block/scope delimiters
`printf`: prints to the terminal
`\n`: newline character
`return`: leave function, return value

Compiling

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Basic Data Types

`char c = 5; char c = 'a';`
one byte, usually for characters (1970: ASCII is fine)
`int i = 5; int i = 0xf; int i = 'a';`
usually 4 bytes, holds integers
`float f = 5; float f = 5.5;`
4 bytes floating point number
`double d = 5.19562`
8 bytes double precision floating point number

Basic Data Types — logic

`int i = 5 / 2; //i = 2`
integer logic, no rounding
`float f = 5.0f / 2; //f = 2.5f`
decimal logic for float and double
`char a = 'a'/ 2 //a = 97 / 2 = 48`
`char` interpreted as character by console

Basic Data Types — signed/unsigned

`signed int i = -5 //i = -5` (two's complement)
`unsigned int i = -5 //i = 4294967291`

Basic Data Types — short/long

`short int i = 1024 //-32768...32767`
`long int i = 1024 //-2147483648...2147483647`

Basic Data Types — more size stuff

`sizeof int; sizeof long int; //4; 4;` (x86 32-Bit)
use data types from `inttypes.h` to be sure about sizes:

```
#include <inttypes.h>
int8_t i; uint32_t j;
```

Basic Data Types — const/volatile

`const int c = 5;`
`i` is constant, changing it will raise compiler error
`volatile int i = 5;`
`i` is volatile, may be modified elsewhere (by different program in shared memory, important for CPU caches, register, assumptions thereof)

Variables — local vs. global

```
int m; // global variable

int myroutine(int j) {
  int i = 5 // local variable
  i = i+j;
  return i;
}
```

global variables (`int m`):
lifetime: while program runs
placed on pre-defined place in memory
basic block/function-local variables (`int i`):
lifetime: during invocation of routine
placed on stack or in registers

Variables — local vs. static

```
int myroutine(int j) {
  static int i = 5;
  i = i+j;
  return i;
}

k = myroutine(1); // k = 6
k = myroutine(1); // k = 7
```

static function-local variables:
saved like global variables
variable persistent across invocations
lifetime: like global variables

Printing

```
int i = 5; float f = 2.5;
printf("The numbers are i=%d, f=%f", i, f);
```

comprised of format string and arguments
may contain format identifiers (`%d`)
see also `man printf`
special characters: encoded via leading backslash:
`\n` newline
`\t` tab
`\'` single quote
`\"` double quote
`\0` `null`, end of string

Compound data types

`struct`ure: collection of named variables (different types)
`union`: *single* variable that can have multiple types
members accessed via `.` operator

```
struct coordinate {
  int x;
  int y;
}
```

```
union longorfloat {
  long l;
  float f;
}

struct coordinate c;
c.x = 5;
c.y = 6;

union longorfloat lf;
lf.l = 5;
lf.f = 6.192;
```

Functions

encapsulate functionality (*reuse*)
code structuring (*reduce complexity*)
must be **declared** and **defined**
Declaration: states signature
Definition: states implementation (implicitly declares function)

```
int sum(int a, int b); // declaration

int sum(int a, int b) { // definition
  return a+b;
}
```

Header files

header file for frequently used declarations
use `extern` to declare global variables defined elsewhere
use `static` to limit scope to current file (e.g. `static float pi` in `sum.c`: no `pi` in `main.c`)

```
// mymath.h
int sum(int a, int b);
extern float pi;

// sum.c
#include "mymath.h"

float pi = 3.1415927;
int sum(int a, int b) {
  return a+b;
}

// main.c
#include <stdio.h>
#include "mymath.h"

void main() {
  printf("%d\n", sum(1,2));
  printf("%f\n", pi);
}
```

Data Segments and Variables

Stack: local variables
Heap: variables crated at runtime via `malloc()`/`free()`
Data Segment: static/global variables
Code: functions

Function overloading

no function overloading in C!
use arrays ore pointers

Pointers

```
int a = 5;
int *p = &a // points to int, initalized to point to a
int *q = 32 // points to int at address 32
int b = a+1;
int c = *p; // dereference(p) = dereference(&a) = 5
int d = (*p)+2 // = 7
int *r = p+1; // pointing to next element p is pointing to
int e = *(p+2) // dereference (p+2) = d = 7
```

Pointers — linked list

linked-list implementation via next-pointer

```
struct ll {
  int item;
  struct ll *next;
}

struct l first;
first.item = 123;

struct ll second;
second.item = 456;
first.next = &second;
```

Arrays

= fixed number of variables *continuously laid out in memory*

```
int A[5]; // declare array (reserve memory space)
A[4] = 25; A[0] = 24; // assign 25 to last, 24 to first elem
char c[] = {'a',5,6,7,'B'} // init array, length implicit
c[64] = 'Z' // NO bounds checking at compile/run (may raise protection
       fault)

// declare pointer to array; address elements via pointer:
char *p = c;
*(p+1) = 'Z'; p[3] = 'B'; char b = *p; // = 'a'
```

Strings

= array of `char`s terminated by `NULL`:

```
char A[] = { 'T', 'e', 's', 't', '\0' };
char A[] = "Test";
```

declaration via pointer:
`const char *p = "Test";`
common string functions (`string.h`):
- length: `size_t strnlen(const char *s, size_t maxlen)`
- compare:
  `int strncmp(const char *s1, const char *s2, size_t n);`
- copy: `int strncpy(char *dest, const char *src size_t n);`
- tokenize: `char *strtok(char *str, const char *delim);`
  (e.g. split line into words)

Arithmetic/bitwise operators

arithmetic operators:
`a+b`, `a++`, `++a`, `a+=b`, `a-b`, `a--`, `--a`, `a-=b`, `a*b`, `a*=b`, `a/b`, `a/=b`, `a%b`, `a%=b`
logical operators:
`a&b`, `a|b`, `a>>b`, `a<<b`, `a^b`, `~a`
difference pre-/post-increment:

```
int a = 5;
if(a++ == 5) printf("Yes"); // Yes
a = 5;
if(++a == 5) printf("Yes"); // nothing
```

operators in order of precedence:

```
(), [], ->, .
!, ++, --, +y, -y, *z, &=, (type), sizeof
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
?, :
=, +=, -=, *=, /=, %=, &=, ~=,=, «=, »=|
,
```

Structures

brackets only needed for multiple statements
`if`/`else`, `for`, `while`, `do-while`, `switch`
may use `break`/`continue`
switch: need break statement, otherwise will fall through

```
if(a==b) printf("Equal") else printf("Different");
for(i=10; i>=10; i--) printf("%d", i+1);
int i=10; while(i--) printf("foo");
int i=0; do printf("bar"); while(i++ != 0);

char a = read();
switch(a) {
  case '1':
    handle_1();
    break;
  default:
    handle_other();
    break;
}
```

Type casting

explicit casting: precision loss possible
  `int i = 5; float f = (float)i;`
implicit casting: if no precision is lost
  `char c = 5; int i = c;`
pointer casting: changes address calculation
  `int i = 5; char *p = (char *)&i; *(p+1)= 5;`
type hierarchy: „wider"/„shorter" types
  `unsigned int` wider than `signed int`
  operators cast parameters to widest type
  Attention: assignment cast after operator cast

C Preprocessor

modifies *source code* before compilation
based on preprocessor *directives* (usually starting with `#`)
`#include <stdio.h>`, `#include "mystdio.h"`:
  copies contents of file to current file
only works with strings in source file
completely ignores C semantics

Preprocessor — search paths

`#include <file>`: system include, searches in:
  `/usr/local/include`
  `libdir/gcc/[target]/[version]/include`
  `/usr/[target]/include`
  `/usr/include`
  (target: arch-specific (e.g. i686-linux-gnu),
   version: gcc version (e.g. 4.2.4))
`#include "file"`: local include, searches in:
  directory containing current file
  then paths specified by `-i <dir>`
  then in system include paths

Preprocessor — definitions

defines introduce replacement strings (can have arguments, based on string replacement)
can help code structuring, often leading to source code cluttering

```
#define PI 3.14159265
#define TRUE (1)
#define max(a,b) ((a > b) ? (a) (b))
#define panic(str) do { printf(str); for (;;) } while(0);

#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#endif
```

Preprocessor — predefined macros

system-specific:
  `__unix__`, `_WIN32`, `__STDC_VERSION__`
useful:
  `__LINE__`, `__FILE__`, `__DATE__`

Libraries

= collection of functions contained in object files, glued together in dynamic/static library
ex.: Math header contains declarations, but not all definitions
  ⇝ need to link math library: `gcc math.c -o math -lm`

```
#include <math.h>
#include <stdio.h>

int main() {
  float f = 0.555f;
  printf("%f", sqrt(f*4));
  return 0;
}
```

Introduction to Operating Systems

What's an OS?

abstraction: provides abstraction for applications
  manages and hides hardware details
  uses low-level interfaces (not available to applications)
  multiplexes hardware to multiple programs (*virtualisation*)
  makes hardware use efficient for applications

protection:

from processes using up all resources (*accounting*, *allocation*)

from processes writing into other processes memory

resource managing:

manages + multiplexes hardware resources

decides between conflicting requests for resource use

strives for efficient + fair resource use

control:

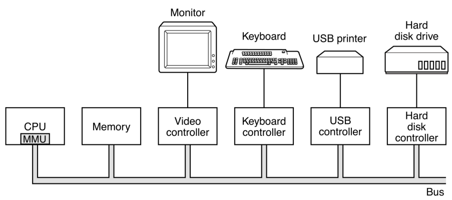
controls program execution

prevents errors and improper computer use

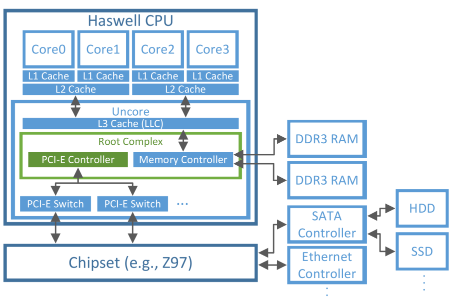
⇝ **no universially accepted definition**

Hardware Overview

CPU(s)/devices/memory (conceptually) connected to common bus
CPU(s)/devices competing for memory cycles/bus
all entities run concurrently



today: multiple busses



Central Processing Unit (CPU) — Operation

fetches instructions from memory, executes them (instruction format/-set depends on CPU)
CPU internal registers store (meta-)data during execution (general purpose registers, floating point registers, instruction pointer (IP), stack pointer (SP), program status word (PSW),...)

execution modes:

user mode (x86: *Ring 3*/*CPL 3*):

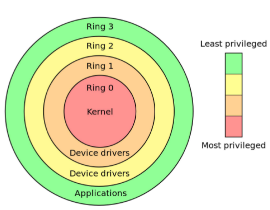
only non-privileged instructions may be executed

cannot manage hardware → **protection**

kernel mode (x86: *Ring 0*/*CPL 0*):

all instructions allowed

can manage hw with **privileged instructions**



Random Access Memory (RAM)

keeps currently executed instructions + data
today: CPUs have built-in *memory controller*
root complex connected directly via

„wire" to caches

pins to RAM

pins to PCI-E switches

Caching

RAM delivers instructions/data slower than CPU can execute
memory references typicalle follow *locality principle*:

spatial locality: future refs often near previous accesses (e.g. next byte in array)

temporal locality: future refs often at previously accessed ref (e.g. loop counter)

caching helps mitigating this memory wall:

copy used information temporarily from slower to faster storage

check faster storage first before going down **memory hierarchy**

if not, data is copied to cache and used from there

Access latency:

register: ∼1 CPU cycle

L1 cache (per core): ∼4 CPU cycles

L2 cache (per core pair): ∼12 CPU cycles

L3 cache/LLC (per uncore): ∼28 CPU cycles (∼25 GiB/s)

DDR3-12800U RAM: ∼28 CPU cycles + ∼ 50ns (∼12 GiB/s)

Caching — Cache Organisation

caches managed in hardware
divided into *cache lines* (usually 64 bytes each, unit at which data is exchanged between hierarchy levels)
often separation of data/instructions in faster caches (e.g. L1, see *harward architecture*)
cache hit: accessed data already in cache (e.g. L2 cache hit)
cache miss: accessed data has to be fetched from lower level
cache miss types:

compulsory miss: first ref miss, data never been accessed

capacity miss: cache not large enough for process working set

conflict miss: cache has still space, but collisions due to placement strategy

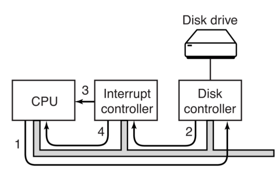
Interplay of CPU and Devices

I/O devices and CPU execute concurrently
Each device controller

- is in charge of particular device
- has local buffer

Workflow:

1. CPU issues commands, moves data to devices
2. Device controller informs APIC (*Advanced Programmable Interrupt Controller*) that operation is finished
3. APIC signals CPU
4. CPU receives device/interrupt number from APIC, executes handler



Device control

Devices controlled through their **device controller**, accepts commands from OS via **device driver**

devices controlled through device registers and device memory:

control device by writing device registers

read status of device by reading device registers

pass data to device by reading/writing device memory

2 ways to access device registers/memory:

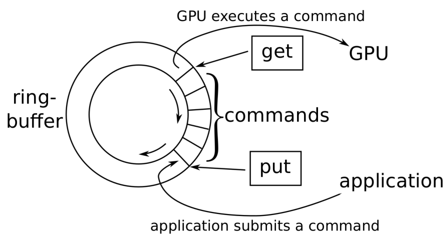
1. **port-mapped IO** (PMIO):
   use special CPU instructions to access port-mapped registers/memory
   e.g. x86 has different `in`/`out`-commands that transfer 1,2 or 4 bytes between CPU and device
2. **memory-mapped IO** (MMIO):
   use same address space for RAM and device memory
   some addresses map to RAM, others to different devices
   access device's memory region to access device registers/memory

some devices use hybrid approaches using both

Device control — Nvidia general purpose GPU

memory-mapped ring-buffer and `put`/`get`-device

mapping can be exposed to application ⇝ application can submit commands in user-mode



Summary

The OS is an abstraction layer between applications and hardware (multiplexes hardware, hides hardware details, provides protection between processes/users)

The CPU provides a separation of User and Kernel mode (which are required for an OS to provide protection between applications)

CPU can execute commands faster than memory can deliver instructions/data – memory hierarchy mitigates this memory wall, needs to be carefully managed by OS to minimize slowdowns

device drivers control hardware devices through PMIO/MMIO

Devices can signal the CPU (and through the CPU notify the OS) through interrupts

OS Concepts

OS Invokation

OS Kernel does **not** always run in background!

Occasions invoking kernel, switching to kernel mode:

1. **System calls**: User-Mode processes require higher privileges
2. **Interrupts**: CPU-external device sends signal
3. **Exceptions**: CPU signals unexpected condition

System Calls — Motivation

Problem: protect processes from one another
Idea: Restrict processes by running them in user-mode
⇝ **Problem**: now processes cannot manage hardware,...
who can switch between processes?
who decides if process may open certain file?
⇝ **Idea**: OS provides **services** to apps
app calls system if service is needed (**syscall**)
OS checks if app is allowed to perform action
if app may perform action and hasn't exceeded quota, OS performs action in behalf of app in kernel mode

System Calls — Examples

`fd = open(file, how,...)` – open file for read/write/both
documented e.g. in `man 2 write`
overview in `man 2 syscalls`

System Calls vs. APIs

syscalls: interface between apps and OS services, limited number of well-defined entry points to kernel
APIs: often used by programmers to make syscalls
e.g. `printf` library call uses `write` syscall
common APIs: Win32, POSIX, C API

System Calls — Implementation

trap instruction: single syscall interface (entry point) to kernel
switches CPU to kernel mode, enters kernel in same, predefined way for all syscalls
system call dispatches then acts as syscall multiplexer
syscalls identified by number passed to trap instruction
syscall table maps syscall numers to kernel functions
dispatcher decides where to jump based on number and table
programs (e.g. `stdlib`) have syscall number compiled in!
⇝ never reuse old numbers in future kernel versions

Interrupts

devices use interrupts to signal predefined conditions to OS
reminder: device has „interrupt line" to CPU
e.g. device controller informs CPU that operation is finished
programmable interrupt controller manages interrupts
interrupts can be **masked**
masked interrupts: queued, delivered when interrupt unmasked
queue has finite length ⇝ interrupts can get lost
noteable interrupt examples:

1. *timer-interrupt*: periodically interrupts processes, switches to kernel ⇝ can then switch to different processes for fairness
2. *network interface card* interrupts CPU when packet was received ⇝ can deliver packet to process and free NIC buffer

when interrupted, CPU

1. looks up **interrupt vector** (= table pinned in memory, contains addresses of all service routines)
2. transfers control to respective **interrupt service routine** in OS that handles interrupt

interrupt service routine must first save interrupted processe's state (instruction pointer, stack pointer, status word)

Exceptions

sometimes unusual condition makes it impossible for CPU to continue processing
⇝ **Exception** generated within CPU:

1. CPU interrupts program, gives kernel control
2. kernel determines reason for exception
3. if kernel can resolve problem ⇝ does so, continues **faulting instruction**
4. kills process if not

Difference to Interrupts: interrupts can happen in any context, exceptions always occur asynchronous and in process context

OS Concepts — Physical Memory

up to early 60s:

- programs loaded and run directly in *physical memory*
- program too large $\longrightarrow$ partitioned manually into *overlays*
- OS then swaps overlays between disk and memory
- different jobs could obeserve/modify eachother

OS Concepts — Address Spaces

bad programs/people need to be isolated
Idea: give every job the illusion of having all memory to itself
every job has own *address space*, can't name addresses of others
jobs always and only use virtual addresses

Virtual Memory — indirect addressing

Today: every CPU has built-in **memory management unit** (*MMU*)
MMU translates virtual addresses to physical addresses at every store/load operation
$\rightsquigarrow$ address translation protects one program from another
Definitions:
Virtual address: address in process' address space
Physical address: address of real memory

Virtual Memory — memory protection

MMU allows kernel-only virtual addresses
- kernel typically part of all address spaces
- ensures that apps can't touch kernel memory

MMU can enforce *read-only* virtual addresses
- allows safe sharing of memory between apps

MMU can enforce execute disable
- makes code injection attacks harder

Virtual Memory — page faults

not all addresses need to be mapped at all times
- MMU issues *page fault* exception when accessed virtual address isn't mapped
- OS handles page faults by loading faulting addresses and then continuing the program
- $\rightsquigarrow$ memory can be **over-committed**: more memory than physically available can be allocated to application

page faults also issued by MMU on illegal memory accesses

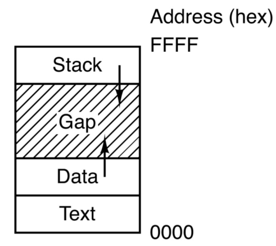
OS Concepts — Processes

= program in execution („instance" of program)
each process is associated with a **process control block** (*PCB*)
contains information about allocated resources
each process is associated with a virtual **address space** (*AS*)
- all (virtual) memory locations a program can name
- starts at 0 and runs up to a maximum
- address 123 in AS1 generally $\neq$ address 123 in AS2
- indirect addressing $\rightsquigarrow$ different ASes to different programs
- $\rightsquigarrow$ protection between processes

OS Concepts — address space layout

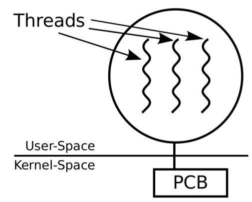
address spaces typically laid-out in different **sections**
- memory addresses between sections **illegal**
- illegal addresses $\rightsquigarrow$ page fault
- more specifically calls **segmentation fault**
- OS usually kills process causing segmentation fault

Stack: function history, local variables
Data: Constants, static/global variables, strings
Text: Program code



OS Concepts — Threads

each progress: ≥ 1 threads (representing execution states)
IP stores currently executed instruction (address in `text` section)
SP register stores address of stack top
(> 1 threads $\longrightarrow$ multiple stacks!)
PSW contains flags about execution history
(e.g. last calculation was 0 $\longrightarrow$ used in following jump instruction)
more general purpose registers, floating point registers,...

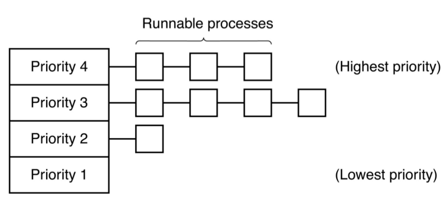


OS Concepts — Policies vs. Mechanisms

separation useful when designing OS
Mechanism: implementation of what is done
(e.g. commands to put a HDD into standby mode)
Policy: rules which decide when what is done and how much
(e.g. how often, how many resources are used,...)
$\longrightarrow$ *mechanismes can be reused even when policy changes*

OS Concepts — Scheduling

multiple processes/threads available $\rightsquigarrow$ OS needs to switch between them (for multitasking)
scheduler decides which job to run next (policy)
dispatcher performs task-switching (mechanism)
schedulers try to
- provide fairness
- while meeting goals
- and adhering to priorities



OS Concepts — Files

OS hides peculiarities of disks,...
programmer uses device-independent *files*/*directories* for persistent storage
Files: associate *file name* and *offset* with bytes
Directories: associate *directory names* with directory names or file names
File System: ordered block collection
- main task: translate (dir name + file name + offset) to block
- programmer uses file system operations to operate on files (`open`, `read`, `seek`)

processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts — Directory Tree

directories form *directory tree*/*file hierarchy*
⟶ structure data
root directory: topmost directory in tree
files specified by providing *path name* to file

OS Concepts — Mounting

Unix: common to orchestrate multiple file systems in single file hierarchy
file systems can be *mounted* on directory
Win: manage multiple directory hierarchies with drive letters
(e.g. `C:\Users`)

OS Concepts — Storage Management

OS provides uniform view of information storage to file systems
- *drivers* hide specific hardware devices
  ⟶ hides device peculiarities
- general interface abstracts physical properties to logical units
  ⟶ block

OS increases I/O performance:
- **Buffering**: Store data temporarily while transferred
- **Caching**: Store data parts in faster storage
- **Spooling**: Overlap one job's output with other job's input

Processes

The Process Abstraction

computers do „several things at the same time" (just looks this way though quick process switching (**Multiprogramming**))
⇝ **process** abstraction models this concurrency:
- container contains information about program execution
- conceptually, every progress has own „virtual CPU"
- execution context is changed on process switch
- dispatcher switches context when switching processes
- **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogon

Program/Process like Recipe/Cooking
Recipe: lists ingredients, gives algorithm what to do when
⇝ program describes memory layout/CPU instructions
Cooking: activity of using the recipe
⇝ process is activity of executing a program
multiple similar recipes for same dish
⇝ multiple programs may solve same problem
recipe can be cooked in different kitchens at the same time
⇝ program can be run on different CPUs at the same time
(as different processes)
multiple people can cook one recipe
⇝ one process can have several worker threads

Concurrency vs. Parallelism

OS uses currency + parallelism to implement multiprogramming

1. **Concurrency**: multiple processes, one CPU
   ⇝ not at the same time
2. **Paralellism**: multiple processes, multiple CPU
   ⇝ at the same time

Virtual Memory Abstraction — Address Spaces

every process has own *virtual addreses* (vaddr)
MMU relocates each load/store to *physical memory* (pmem)
processes never see physical memory, can't access it directly
\+ MMU can enforce protection (mappings in kernel mode)
\+ programs can see more memory than available
80:20 rule: 80% of process memory idle, 20% active
can keep working set in RAM, rest on disk
– need special MMU hardware

Address Space (Process View)

code/data/state need to be organized within process
⇝ **address space layout**
Data types:

1. *fixed size* data items
2. data naturally *free'd in reverse allocation order*
3. data *allocated/free'd „randomly"*

compiler/architecture determine how large int is and what instructions are used in text section (`code`)
Loader determines based on exe file how executed program is placed in memory

Segments — Fixed-Size Data + Code

some data in programs never changes or will be written but never grows/shrinks
⇝ memory can be statically allocated on process creation
BSS segment (*block started by symbol*):
- statically allocated variables/non-initialized variables
- executable file typically contains starting address + size of BSS
- entire segment initially 0

Data segment:
- fixed-size, initlized data elements (e.g. global variables)

read-only data segment:
- constant numbers, strings

All three sometinmes summarized as one segment
compiler and OS decide ultimately where to place which data/how many segments exist

Segments — Stack

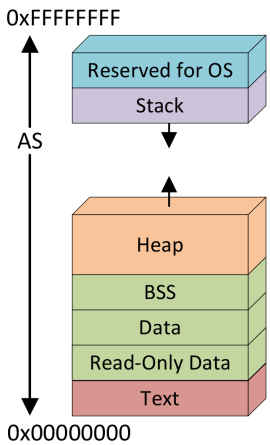
some data naturally free'd in reverse allocation order
⇝ very easy memory management (stack grows upwards)
fixed segment starting point
store top of latest allocation in **stack pointer** (SP)
(initialized to starting point)
allocate `a` byte data structure: `SP += a; return(SP - a)`
free `a` byte data structure: `SP -= a`

Segments — Heap (Dynamic Memory Allocation)

some data „randomly" allocated/free'd
two-tier memory allocation:

1. allocate large memory chunk (**heap segment**) from OS
   - base address + **break pointer** (BRK)
   - process can get more/give back memory from/to OS

2. dynamically partition chunk into smaller allocations
   - `malloc`/`free` can be used in random oder
   - purely user-space, no need to contact kernel



Summary

recipe vs. cooking is like program vs. process

processes = ressource container for OS

process feels alone: has own CPU + memory

OS implements multiprogramming through rapid process switching

Process API

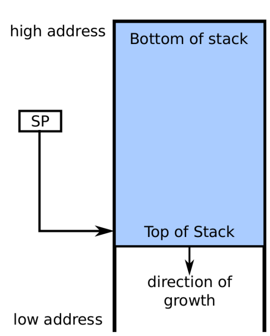
Execution Model — Assembler (simplified)

OS interacts directly with compiled programs
- switch between processes/threads ⇝ **save**/**restore** state
- deal with/pass on **signals**/**exceptions**
- receive **requests** from applications

Instructions:
- `mov`: Copy referenced data from second operand to first operand
- `add`/`sub`/`mul`/`div`: Add,...from second operand to first operand
- `inc`/`dec`: increment/decrement register/memory location
- `shl`/`shr`: shift first operand left/right by amount given by second operand
- `and`/`or`/`xor`: calculate bitwise and,...of two operands storing result in first
- `not`: bitwise negate operand

Execution Model — Stack (x86)

stack pointer (SP): holds address of stack top (growing downwards)
stack frames: larger stack chunks
base pointer (BP): used to organize stack frames



Execution Model — jump/branch/call commands (x86)

`jmp`: continue execution at operand address
`j$condition`: jump depending on PSW content
- true ⇝ jump
- false ⇝ continue
- examples: `je` (jump equal), `jz` (jump zero)

`call`: push function to stack and jump to it
`return`: return from function (jump to return address)

Execution Model — Application Binary Interface (ABI)

standardizes binary interface between programs, modules, OS:
- executable/object file layout
- calling conventions
- alignment rules

calling conventions: standardize exact way function calls are implemented
- ⇝ interoperability between compilers

Execution Model — calling conventions (x86)

function call (caller):

1. save local scope state
2. set up parameters where function can find them
3. transfer control flow

function call (called function):

1. set up new local scope (local variables)
2. perform duty
3. put return value where caller can find it
4. jump back to caller (IP)

Passing parameters to the system

parameters are passed through **system calls**
call number + specific parameters must be passed
parameters can be transferred through
- **CPU registers** (~6)
- **Main Memory** (heap/stack – more parameters, data types)

ABI specifies how to pass parameters
return code needs to be returned to application
- **negative**: error code
- **positive + 0**: success
- usually returned via A+D registers

System call handler

implements the actual serivce called through a syscall:

1. saves tainted registers
2. reads passed parameters
3. sanitizes/checks parameters
4. checks if caller has enough permissions to perform the requested action
5. performs requested action in behalf of the caller
6. returns to caller with success/error code

Process API — creation

process creation events:

1. system initialization
2. process creation syscall
3. user requests process creation
4. batch job-initiation

events map to two mechanisms:

1. Kernel spawns initial user space process on boot (Linux: `init`)
2. User space processes can spawn other processes (within their quota)

Process API — creation (POSIX)

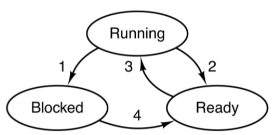
`PID`: identifies process
`pid = fork()`: duplicates current process:
  returns 0 to new child
  returns new `PID` to parent
  ⇝ child and parent independent after `fork`
`exec(name)`: replaces own memory based on executable file
  `name` specifies binary executable file
`exit(status)`: terminates process, returns `status`
`pid = waitpid(pid, &status)`: wait for child termination
  - `pid`: process to wait for
  - `status`: points to data structure that returns information about the process
    (e.g., exit status)
  - passed `pid` is returned on success, `-1` otherwise
process tree: processes create child processes, which create child processes, …
  - parent and child execute concurrently
  - parent waits for child to terminate (collecting the exit state)

Daemons

= program designed to run in the background
detached from parent process after creation, reattached to process tree root (`init`)

Process States

blocking: process does nothing but wait
  - usually happens on syscalls (OS doesn't run process until event happens)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process Termination

different termination events:

1. normal exit (voluntary)
   - `return 0` at end of `main`
   - `exit(0)`
2. error exit (voluntary)
   - `return x` ($x \neq 0$) at end of `main`
   - `exit(x)` ($x \neq 0$)
   - `abort()`
3. fatal error (involuntary)
   - OS kills process after exception
   - process exceeds allowed ressources
4. killed by another process (involuntary)
   - another process sends kill signal (only as parent process or administrator)

Exit Status

voluntary exit: process returns exit status (integer)
ressources not completely free'd after process terminates
  ⇝ **Zombie** or **process stub** (contains exit status until collected via `waitpid`)
Orphans: Processes without parents
  - usually adopted by `init`
  - some systems kill all children when parent is killed
exit status on involuntary exit:
  - Bits 0-6: signal number that killed process (0 on normal exit)
  - Bit 7: set if process was killed by signal
  - Bits 8-15: 0 if killed by signal (exit status on normal exit)

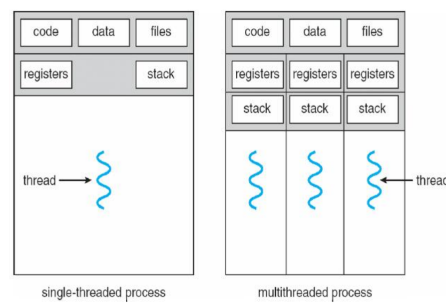
Threads

Processes vs. Threads

Traditional OS: each process has
  - own address space
  - own set of allocated resources
  - *one* thread of execution (= one execution state)
Modern OS: processes + threads of execution handled more flexibly
  - *processes* provide abstraction of address space and resources
  - *threads* provide abstraction of execution states of that address space
Exceptions:
  - sometimes different threads have different address spaces
  - *Linux*: threads = regular processes with shared resources and AS regions

Threads — why?

many programs do multiple things at once (e.g. web server)
  ⇝ writing program as many sequential threads may be easier than with blocking operations
Processes: rarely share data (if, then explicitly)
Threads: closely related, share data



Threads — POSIX

`PThread`: base object with
  - *identifier* (thread ID, TID)
  - *register set* (including IP and SP)
  - *stack area* to hold execution state
`Pthread_create`: create new thread
  - Pass: *pointer* to `pthread_t` (will hold TID after successful call)
  - Pass: *attributes*, *start function*, *arguments*
  - Returns: 0 on success, error value else
`Pthread_exit`: terminate calling thread
  - Pass: exit code (casted to void pointer)
  - Free's resources (e.g. stack)
`Pthread_join`: wait for specified thread to exit
  - Pass: `ptread_t` to wait for (or `-1` for any thread)
  - Pass: pointer to pointer for exit code
  - Returns: 0 on success, error value else
`Pthread_yield`: release CPU to let another thread run

Threads — Problems

Processes vs. Threads:
  - *Processes*: only share resources explicitly
  - *Threads*: more shared state ⟶ more can go wrong

Challenges: programmer needs to take care of
  - *activities*: dividing, ordering, balancing
  - *data*: dividing
  - *shared data*: access synchronizing

PCB vs. TCP

PCB (*process control block*): information needed to implement processes

- always known to OS

TCB (*thread control block*): per thread data

- OS knowledge depends on *thread model*

PCB	TCB
Address space	Instruction pointer
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

Thread models

Kernel Thread: known to OS kernel
User Thread: known to process
N:1-Model: kernel only knows one of possibly multiple threads

- N:1 user threads = *user level threads* (ULT)

1:1-Model: each user thread maps to one kernel thread

- 1:1 user threads = *kernel level threads* (KLT)

M:N-Model (hybrid model): flexible mapping of user threads to less kernel threads

Thread models — N:1

Kernel only manages process $\longrightarrow$ multiple threads unknown to kernel
Threads managed in user-space library (e.g. GNU Portable Threads)
Pro:

+ faster thread management operations (up to 100 times)
+ flexible scheduling policy
+ few system resources
+ useable even if OS doesn't support threads

Con:

- no parallel execution
- whole process blocks if one user thread blocks
- reimplementing OS parts (e.g. scheduler)

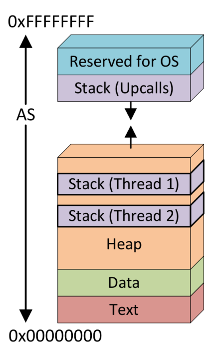
Stack:

- main stack known to OS used by thread library
- own execution state (= stack) dynamically allocated by user thread library for each thread
- possibly own stack for each exception handler

Heap:

- concurrent heap use possible
- *Attention*: not all heaps are reentrant

Data: divided into BSS, data and read-only data here as well



Thread models — 1:1

kernel knows + manages every thread
Pros:

+ real parallelism possible
+ threads block individually

Cons:

- OS manages every thread in system (TCB, stacks,…)
- Syscalls needed for thread management
- scheduling fixed in OS

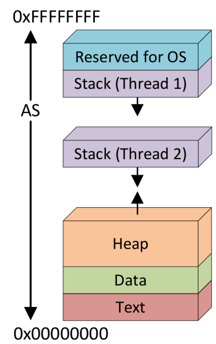
Stack:

- own execution state (= stack) for every thread
- possibly own stack for (each) exception handler

Heap:

- parallel heap use possible
- *Attention*: not all heaps are thread-safe
- if thread-safe: not all heap implementations perform well with many threads

Data: divided into BSS, data and read-only data here as well



Thread models — M:N

Principle: M ULTs are mapps to (at most) N KLT

- *Goal*: pros of ULT and KLT — non-blocking with quick management
- create sufficient number of KLTs and flexibly allocate ULTs to them
- *Idea*: if ULT blocks ULTs can be switched in userspace

Pros:

+ flexible scheduling policy
+ efficient execution

Cons:

- hard to debug
- hard to implement (e.g. blocking, number of KLTs,…)

Implementation — Upcalls:

- kernel notices that thread will block $\longrightarrow$ sends signal to process
- upcall notifies process of thread id and event that happened
- exception handler of process schedules a different process thread
- kernel later informs process that blocking event finished via other upcall

Scheduling

Motivation

K jobs ready to run, $K > N \geq 1$ CPUs available
Scheduling Problem:

- Which jobs should kernel assign to which CPUs?
- When should it make decision?

Dispatcher

Dispatcher: performs actual process switch

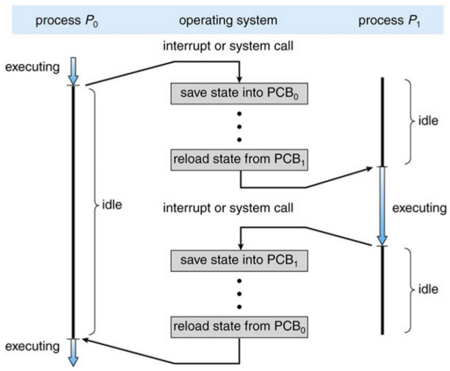
- mechanism
- save/restore process context
- switch to user mode

Scheduler: selects next process to run based on *policy*

Voluntary Yielding vs. Preemption

kernel responsible for CPU switch
kernel doesn't always run $\longrightarrow$ can only dispatch different process when invoked
cooperative multitasking: running process performs *yield* syscall $\longrightarrow$ kernel switches process

preemptive scheduling:
- kernel invoked in certain time intervals
- kernel makes scheduling decisions after every time-slice



Scheduling — Process States

new: process was created but did not run yet
running: instructions are currently being executed
waiting: process is waiting for some event
ready: process is waiting to by assigned a processor
terminated: process has finished execution

Scheduling — long-term vs. short-term

Short-term scheduler (CPU Scheduler, focused on in this lecture):
- selects process to run next, allocates CPU
- invoked frequently (ms) ⇝ must be fast

Long-term scheduler (job scheduler):
- selects process to be brought into ready queue
- invoked very infrequently (s, m) ⇝ can be slow
- controls degree of *multiprogramming*

Scheduling queues

job queue: set of all processes in system
ready queue: process in main memory, ready or waiting
device queue: processes waiting for I/O device

Scheduling Policies — Categories

batch scheduling:
- still widespread in business (payroll, inventory,…)
- no users waiting for quick response
- non-preemptive algorithms acceptable $\rightarrow$ less switches $\rightarrow$ less overhead

interactive scheduling:
- need to optimize for response time
- preemption essential to keep processes from hogging CPU

real-time scheduling:
- guarantee job completion within time constraints
- need to be able to plan when which process runs + how long
- preemption not always needed

Scheduling Policies — Goals

General:
- *fairness*: give each process fair share of CPU
- *balance*: keep all parts of system busy

batch scheduling:
- *throughput*: number of processes that complete per time unit
- *turnaround time*: time from job submission to job completion
- *CPU utilization*: keep CPU as busy as prossible

interactive scheduling:
- *waiting time*: reduce time a process waits in waiting queue
- *response time*: time from request to first response

real-time scheduling:
- *meeting deadlines*: finishing jobs in time
- *predictability*: minimize jitter

Scheduling Policies — first come first served

intuitively clear
Example: 3 processes arrive at time 0 in the order P_1, P_2, P_3

Process	Burst time	Turnaround time
P_1	24	24
P_2	3	27
P_3	3	30

⇝ average turnaround time 27 $\rightarrow$ *can we do better?*
Conclusion: if processes would arrive in order P_2, P_3, P_1, average turnaround time would be 13
⇝ *good scheduling can reduce turnaround time*

Scheduling Policies — shortest job first

Benefits: optimal average turnaround/waiting/response time
Challenge: cannot know job lengths in advance
Solution: predict length of next CPU burst for each process
⇝ schedule process with shortest burst next
Burst Estimation: *exponential averaging*
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
(t_n: actual length of n-th CPU burst, τ_{n+1}: predicted length of next CPU burst, $0 \leq \alpha \leq 1$)

Process Behaviour — CPU bursts

CPU bursts exists because processes wait for I/O
CPU-bound processes: spends more time doing computations
⇝ few very long CPU bursts
I/O-bound processes: spends more time doing I/O
⇝ many short CPU bursts

Scheduling Policies — preemptive shortest-job-first

SJF optimizes waiting/response time
⇝ what about throughput?
Problem: CPU-bound jobs hold CPU until exit or I/O $\rightarrow$ *poor I/O utilization*
Idea: SJF, but preempt periodically to make new scheduling decision
- each time slice: schedule job with shortest remaining time next
- alternatively: schedule job with shortest next CPU burst

Scheduling Policies — round robin

Problem: batch schedulers suffer from starvation and don't provide fairness
Idea: each process runs for small CPU time unit
- *time quantum*/*time slice* length: usually 10-100ms
- preempt processes that have not blocked by end of time slice
- append current thread to end of run queue, run next thread

Caution: time slice length needs to balance interactivity and overhead!
$\rightarrow$ if time slice length in the area of dispatch time, 50% of CPU time wasted for process switching

Scheduling Policies — virtual round robin

Problem: RR is unfair for I/O-bound jobs: they block before using up time quantum
Idea: put jobs that didn't use up their quantum in additional queue
- store share of unused time-slice
- give those jobs additional queue priority
- put them back into normal queue afterwards

Scheduling Policies — (strict) priority scheduling

Problem: not all jobs are equally important
$\rightsquigarrow$ different priorities (e.g., 4)
Solution: associate priority number with each process
- RR for each priority
- *aging*: old low priority processes get executed before new higher priority processes

Scheduling Policies — multi-level feedback queue

Problem: context switching expensive
$\rightsquigarrow$ trade-off between interactivity and overhead?
Goals:
- higher priority for I/O jobs (usually don't use up quantum)
- low priority for CPU jobs (rather run them longer)

Idea: different queues with different priorities and time slice lengths
- schedule queues with (static) priority scheduling
- double time slice lnegth in each next-lower priority
- process to higher priority when they don't use up quantum repetitively
- process to lower priority when they use up quantum repetitively

Scheduling Principles — priority donation

Problem: Process B (higher priority) waits for process A (lower priority)
$\rightsquigarrow$ B has now effectively lower priority
Solution: *priority donation*
- give A priority of B as long as B waits for A
- if C, D, E wait for B $\longrightarrow$ A gets highest priority of B, C, D, E

Scheduling Policies — lottery scheduling

issue number of lottery tickets to processes (amount depending on priority)
amount of tickets controls average proportion of CPU for each process
Scheduling: scheduler draws random number N, process with N-th ticket is executed
processes can transfer tickets to other processes if they wait for them